PORTING CONTIKI TO THE BTNODE PLATFORM

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Cyrill Schluep
2009

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel der Arbeit / Motivation

In heterogenen drahtlosen Sensornetzen werden verschieden Typen von Sensorknoten betrieben, welche auf die unterschiedlichen Bedürfnisse und Aufgaben angepasst sind. Diese Sensorknoten sind über ein Netz von Meshknoten miteinander verbunden. Ein mögliches Szenario wird in [1] beschrieben und in Abbildung 1.1 dargestellt.

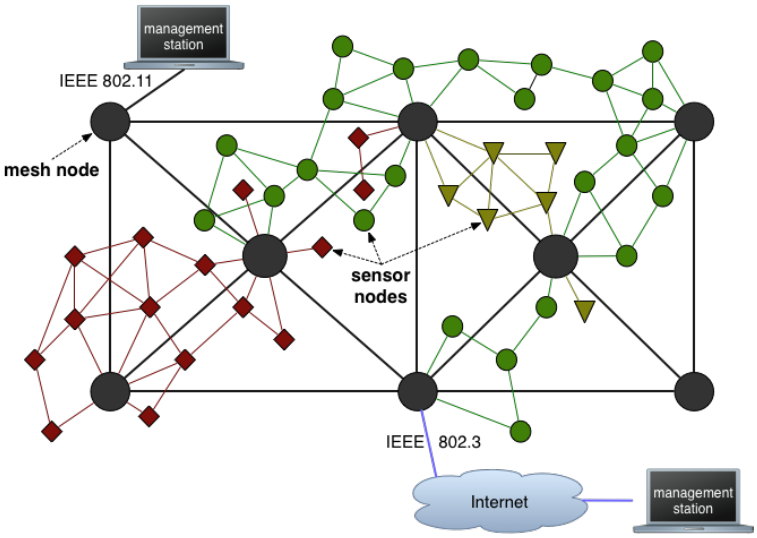

Abbildung 1.1: Heterogenes drahtloses Sensornetzwerk mit Meshknoten

Da die verschiedenen Typen von Sensorknoten keine kompatiblen Radiotransceiver verwenden, kann keine direkte Kommunikation stattfinden. So bilden die Sensorknoten eines Types jeweils ein Subnetz. Diese Subnetze werden nun mithilfe eines Backbones aus Meshknoten miteinander verbunden. Dazu wird ein Sensorknoten über ein serielles Interface an einen Meshknoten angeschlossen. Ebenso kann über die Meshknoten eine Kontrollstation angeschlossen

werden. Mithilfe der Meshknoten kann die Anzahl Hops zwischen zwei kommunizierenden Sensorknoten verringert werden. Zusätzlich ist es von Vorteil, dass alle Sensorknoten mit demselben Betriebssystem ausgestattet sind. Damit wird Kompatibilität innerhalb des Netzes sichergestellt und die Entwicklung von Anwendungen vereinfacht.
Als Betriebssystem wird Contiki (Version 2.x) verwendet. Contiki [2] ist ein modulares, leicht portierbares Betriebssystem, welches vom Swedish Institute of Computer Science entwickelt wurde.
Als Sensorknoten werden neben TMoteSKy und MicaZ, von denen Contiki Portierungen existieren, auch der von der ETH Zürich entwickelte BTnode [3] verwendet. Für ihn existiert bisher das Betriebssystem BTnuts [4] und eine TinyOS [5] Portierung.
Ziel dieser Arbeit ist es, eine funktionierende Contiki Portierung für den BTnode (rev. 3.24) zu erstellen. Dabei sollen grundlegende Hardwarekomponenten sowie die Funkschnittstelle getestet werden und eine Kommunikation zweier Knoten über uIP und/oder RIME möglich sein.

1.2 Aufbau der Arbeit

Zuerst wird die Hardware der BTnodes detailliert beschrieben. Der folgende Abschnitt geht genauer auf das Betriebssystem Contiki ein. Anschliessend werden die notwendigen Schritte einer Portierung aufgeführt und im Detail erläutert. Dabei wird gezielt auf die einzelnen Gerätetreiber eingegangen und Probleme und Schwierigkeiten bei der Portierung analysiert. Am Schluss folgt dann eine Evaluation mit einem Durchsatztest für die Funkschnittstelle.

Kapitel 2

BTnode und Contiki

2.1 Hardware BTnode

Der BTnode ist ein von der ETH Zürich entwickelter Sensorknoten. Er besteht aus folgenden Hardwarekomponenten:

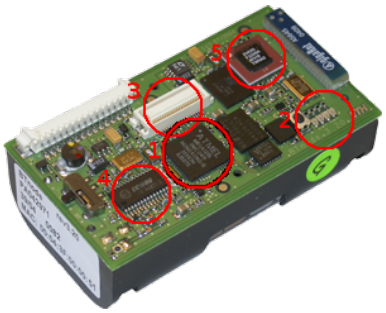

Abbildung 2.1: BTnode

- Mikrokontroller: Atmel ATmega 128L (8 MHz @ 8 MIPS) [6]
- Speicher: 64+180 KByte RAM, 128 KByte FLASH ROM, 4 KByte EEPROM
- Bluetooth: Zeevo ZV4002
- Funk-Controller: Chipcon CC1000 [7]
- Externe Schnittstellen: UART, SPI, 4 LEDs, Reset Knopf
- Externe Stromversorgung oder AA Batterien mit Ein/Aus Switch

Der Mikrokontroller basiert auf der erweiterten RISC Architektur von AVR. Der ATmega128-Prozessor kann mit seinem erweiterten Befehlssatz einen Durchsatz von annähernd 1MIPS pro MHz erreichen. Dies erlaubt die Optimierung der CPU-Geschwindigkeit gegenüber

dem Stromverbrauch.
Der ATmega128 besitzt ausserdem einen intergrierten Flash Speicher von 128Kbyte, 4Kbyte EEPROM, 4Kbyte SRAM, 32 Arbeitsregister, vier verschiedene Timer, zwei serielle Schnittstellen UART, einen Watchdog Timer und eine serielle SPI Schnittstelle. Der Watchdog Timer dient dazu, Endlosschlaufen zur Laufzeit zu erkennen und bei Bedarf den Prozessor neu zu starten.

Der CC1000 Funk-Controller ist für Senden und Empfangen auf den Frequenzen von 315-915 MHz ausgelegt. Dabei ist er bezüglich des Stromverbrauches optimiert. Laut Herstellerbeschreibung eignet sich der CC1000 eher für den Einsatz auf kurzen Distanzen. Dies wird später in der Evaluation bestätigt. Der CC1000 wird über ein serielles Interface konfiguriert. Er unterstützt die Codierungen Manchester und NRZ.
Die SPI Schnittstelle verbindet den Prozessor mit dem CC1000. Über die UART Schnittstelle kann der BTnode an ein externes Gerät (PC) angeschlossen werden.

2.2 Contiki

2.2.1 Übersicht

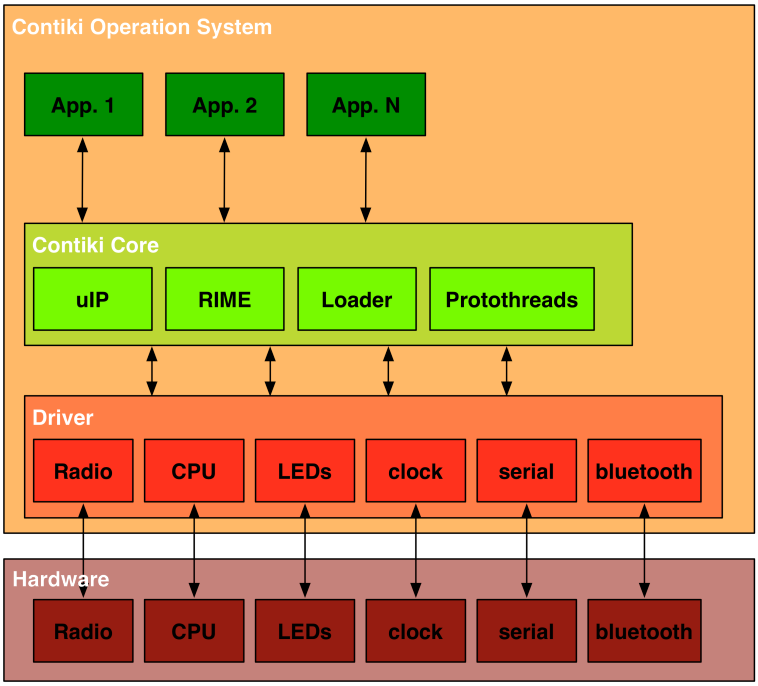

Abbildung 2.2: Aufbau des Contiki Betriebssystems

Das Betriebssystem Contiki wurde vom Swedish Institute of Computer Science entwickelt. Ziel war es, ein modulares, leicht zu portierendes Betriebssystem für drahtlose Sensorknoten

zu schaffen. Die Modularität von Contiki ermöglicht eine gute Portierung und die Entwicklung von hardwareunabhängigen Anwendungen. Der Kern des Contiki Betriebssystems besteht aus den Kommunikationsprotokollen, den Protothreads [8] sowie der Laderoutine.
Die Idee hinter den Protothreads ist, den Programmaufbau in Form von Threads zu gestalten und gleichzeitig eine minime Speicherbelegung zu erzielen. Protothreads besitzen keinen eigenen Stack und benötigen somit nur wenig Speicher. Anwendungen in Contiki sind als Protothreads implementiert.
Als Kommunikationsprotokolle unterstützt Contiki IPv4, IPv6 sowie RIME [9] (siehe Abb. Lit-

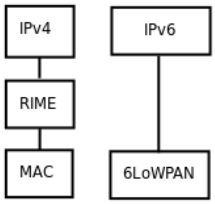

Abbildung 2.3: Kommunikationsstacks

eraturLayer). RIME ist ein Kommunikationsstack für Sensornetzwerke. RIME unterscheidet sich vor allem dadurch von herkömmlichen Kommunikationsstacks, dass die verschiedenen Schichten sehr dünn gehalten werden. RIME vereinfacht die Implementierung von Kommunikationsprotokollen in Sensornetzwerken mit nur sehr geringem Ressourcenverbrauch. IPv4 setzt auf dem RIME Protokoll auf. Dazu ist der uIPv6 Stack ebenso Teil von Contiki. Als MAC-Protokoll wird nullmac angeboten. Das nullmac Protokoll ist eine minimale implementierung eines MAC-Protokolls in Contiki. Es werden lediglich die Funktionen zum initieren, senden und empfangen implementiert. Dabei beschränkt sich nullmac auf das reine Weiterleiten von Paketen.
Die Laderoutine ist verantwortlich das Betriebssystem zu laden. Dies bedeutet die Initiierung aller Treiber und Protokolle und das Starten der Anwendungen.
Der hardwarespezifische Teil von Contiki besteht aus den Gerätetreibern. Jede Komponente benötigt einen Treiber, wobei vorgegeben ist, wie die jeweiligen Treiber anzusprechen sind. Diese Vorgaben finden sich in Form von Funktionsbeschreibungen, welche von einem Treiber implementiert werden müssen.

2.2.2 Contiki als Entwicklungsplattform

Contiki ist vollständig in C geschrieben. Die wichtigsten Verzeichnisse im Quellcode von Contiki sind:

- `cpu/`
- `platform/`

- `projects/`

Der Quellcode für die verschiedenen Mikrokontroller befindet sich im Verzeichnis `cpu/`, in diesem Fall `cpu/avr/`. Jede Portierung von Contiki für eine Plattform befindet sich im Verzeichnis `platform/`. Einige der unterstützten Plattformen sind *TMote Sky, MicaZ, MSP430* und *AVR-Raven*. Weiter gibt es die speziellen Plattformen *native, netsim* und *cooja*. Die Platform `native` bietet die möglichkeit eine Anwendung lokal zu kompilieren und auszuführen. Dies ermöglicht das Testen von Anwendungen in einer virtuellen, architekturunabhängigen Umgebung. Der Vorteil dabei ist, dass so eine Anwendung getestet werden kann, ohne sie auf einen Knoten zu laden. Dabei kann eine Anwendung auch mit einem Debugger analisiert werden. Die Plattform *netsim* bietet zusätzlich noch die Möglichkeit mehrere Sensorknoten mit Netzwerkfunktionen zu simulieren. Die Platform *cooja* ermöglicht dann komplexe Simulationen in einem virtuellen Sensornetzwerk. Die Plattform *native* kann ausserdem bei der Portierung als Vorlage benutzt werden.

Beim Kompilieren einer Anwendung (Anwendung z.b. im Verzeichnis `projects/blink/`) wird mithilfe des Parameters `TARGET=platform` angegeben für welche Plattform der Quellcode kompiliert werden soll.

Kapitel 3

Portierung

3.1 Übersicht des zu portierenden Quellcodes

Beim Portierungsprozess geht es darum, Contiki so zu erweitern, dass es für den BTnode kompiliert und darauf ausgeführt werden kann. Dazu wird eine neue Plattform *btnode* in Contiki erstellt wobei als Basis die Plattform *native* verwendet wird. Folgende Komponenten müssen erstellt werden:

- `contiki-main.c` und `contiki-conf.h` als Main-Funktion und zentrale Konfigurationsdatei
- Makefiles
- Interne Uhr, auf die der Contiki Timer zugreifen kann
- CC1000 Funk-Controller
- LATCH Treiber, wird benutzt, um den integrierten LATCH zu steuern. An diesem sind beispielsweise die LEDs, aber auch on/off für Bluetooth und Funk-Controller gekoppelt.
- LEDs, Funktionen zum Ein-/Ausschalten und Farbdefinitionen
- Serielle Schnittstelle RS232 um Konsolenzugriff und Ausgabe zu ermöglichen

3.2 Contiki

Die Datei `contiki-main.c` beinhaltet die Main-Funktion des Betriebssystems. Hier werden die notwendigen Treiber initialisiert sowie sämtliche Prozesse gestartet. In `contiki-conf.h` stehen alle wichtigen Konstanten die plattformspezifisch sind (wie z.b. die Farben der LEDs).

Contiki hat ein zentrales Makefile, welches beim Kompilieren das plattformspezifische Makefile einbindet. In diesem Makefile werden die Targets sowie die benötigten Treiber definiert. Die wichtigsten Targets sind:

- `make .hex`
  Kompiliert eine Anwendung und erstellt eine Rohdatei, welche geladen werden kann.

- `make .upload`
  Wird benutzt, um ein Programm auf den Sensorknoten zu laden.

- `make login`
  Stellt eine serielle Verbindung zum Sensorknoten her.

- `restart`
  Kann einen Sensorknoten (auf dem ein entsprechendes Programm bereits installiert ist) neu starten.

3.3 Mikrocontroller

Der bereits existierende Quellcode für den Atmel ATmega128 Mikroprozessor wird eingebunden. In der Plattformspezifischen Konfigurationsdatei `contiki-conf.h` muss der Prozessortyp und die Taktfrequenz des Prozessor definiert werden. In `cpu/avr/` werden die Register sowie die Anschlusspins des Prozessors definiert. Weiter werden Funktionen bereitgestellt, um auf den EEPROM und den Flashspeicher zuzugreifen.
Ebenfalls werden hier auch die Treiber der verschiedenen seriellen Anschlüsse definiert (RS232 und SPI). Weiter enthalten sind auch einfache Treiber für die Systemuhr und die LEDs. Von diesen wird hier jedoch nur der RS232 Treiber verwendet, die übrigen werden entweder neu geschrieben oder müssen zumindest stark angepasst werden.

3.4 Gerätetreiber

Die Gerätetreiber werden entweder neu implementiert oder es wird eine architekturspezifische Datei modifiziert. In jedem Fall existieren vorgegebene Funktionen, welche implementiert werden müssen.

3.4.1 Systemuhr

Es wird die interne 8Bit Uhr verwendet. Durch einen Prescaler wird die Uhr so eingestellt, dass alle 1024 Prozessortakte eine Iteration stattfindet. Der Prescaler ist eine interne Schaltung welche den Prozessortakt durch einen bestimmten Wert dividiert. Die in diesem Fall an die Uhr weitergeleitete Frequenz ist also tiefer als die Tatsächliche Taktfrequenz des Prozessors. Bei einem Takt von 7,3728MHz ergeben sich somit 7200 Iterationen pro Sekunde. Mit dem 8Bit Register teilen wir die Zeit nun so auf, dass wir 32 x 225 Iterationen zählen. Dies kann beliebig geändert werden, indem einerseits der Prescaler vermindert wird und gleichzeitig die Anzahl Iterationen pro Sekunde um denselben Faktor erhöht werden. So kann die Genauigkeit im Extremfall bis auf die Taktfrequenz gesteigert werden. Obwohl so natürlich eine höhere Genauigkeit erreicht werden kann, muss hier beachtet werden dass zu häufige Interrupts das System beträchtlich verlangsamen können.

3.4.2 CC1000 Funk-Controller

Die Implementierung des Treibers für den CC1000 ist komplexer und wird deshalb ausführlicher behandelt. In einem ersten Schritt muss der CC1000 nach dem Einschalten konfiguriert werden. Dabei werden ihm über einen Konfigurationsanschluss (die PINs: PALE, PDATA, PCLK in Abbildung 3.1) nacheinander Registernummer und Wert übergeben. Auch wenn es nicht notwendig ist, so wird doch empfohlen den CC1000 nach jedem Einschalten zu konfigurieren. Dazu werden die Standardkonfigurationswerte und die Konfigurationsroutine aus dem Treiber von BTNuts verwendet. Diese haben sich nach eingehender Analyse als sinnvoll erwiesen.

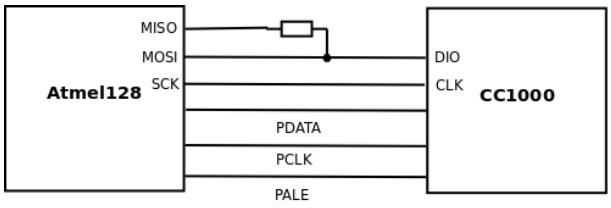

Abbildung 3.1: CC1000 Interface

Der CC1000 kann entweder senden oder empfangen. Es muss also zwischen Sende- und Empfangs-Modus umgeschaltet werden. Einmal konfiguriert wird das Funk-Controller über eine SPI Verbindung (Bestehend aus MISO, MOSI, SCK und DIO, CLK in Abbildung 3.1) angesprochen. Dies bedeutet, dass wir jeweils ein Byte schreiben oder lesen können. Da das CC1000 intern jedoch bitorientiert ist, muss die Byteausrichtung überprüft werden. Ein Empfänger kann nicht wissen, welche 8Bits des empfangenen Bitstroms nun zu einem Byte kombiniert werden sollen. Er erhält alle gesendeten Bits welche jedoch unter Umständen zu anderen Bytes kombiniert werden. In Abbildung 3.2 wird dies verdeutlicht. Oben sind die drei gesendeten Bytes, jeweils als Bitfolge und in hexadezimaler Schreibweise angegeben. Darunter sind die daraus resultierenden möglichen Bytes, die empfangen werden können, dargestellt. Anhand der so empfangenen beiden Bytes kann eindeutig die Verschiebung berechnet werden. Die Verschiebung bleibt innerhalb eines gesendeten Paketes gleich. Nach Sendepausen (also bei einem neuen Paket) muss die Verschiebung jedoch wieder neu berechnet werden.

Das Senden geschieht nach folgendem Muster: Zuerst werden die in Abbildung 3.2 beschriebenen drei Bytes als Erkennungssequenz gesendet. Daraus kann der Empfänger die Verschiebung errechnen. Danach folgt ein Prüfbyte, damit sichergestellt ist, dass die Verschiebung korrekt errechnet wurde (eine falsche Berechnung geschieht beispielsweise bei Bitfehlern im dritten Byte). Nun folgt ein Byte mit der Länge des zu übertragenden Pakets und anschliessend die Nutzdaten. Daraus ergibt sich ein Overhead von insgesamt fünf Bytes bei einer Paketgrösse von bis zu 255 Bytes. Beim Empfangen werden, nachdem die Verschiebung berechnet wurde, immer jeweils zwei aufeinanderfolgene Bytes betrachtet. So erhält man durch Kombination und zurückschieben der betrachteten Bytes das ursprünglich gesendete Byte.

In einer ersten Version des Treibers wird in regelmässigen Abständen ein Byte gelesen. Falls die Erkennungssequenz eintrifft und das Prüfbyte korrekt ist, wird nun so lange gelesen

bis das gesamte Paket empfangen ist. Das Lesen in bestimmten Intervallen und das Warten auf die Erkennungssequenz ist jedoch problematisch, da eine Übertragung eines Byte im Vergleich zum Prozessortakt extrem lange dauert. Somit ist das System lange Zeit durch Warten auf ein Signal blockiert. Ein korrekt funktionierender Treiber ist zwar möglich, bei Ausführung von mehreren Threads sinkt die Leistung jedoch merklich.

Um dieses Problem zu umgehen, wird in dieser Arbeit ein vollständig anderer Ansatz gewählt. Anstatt regelmässig auf eingehende Daten zu warten, wird nun ein Interrupt ausgelöst, sobald eine Übertragung auf dem SPI fertig ist. Dies wird zum Senden und Empfangen gleichermassen benutzt. Wird nun ein Interrupt ausgelöst, wird zuerst entschieden, ob der Sende- oder Empfangsmodus aktiv ist. Danach wird anhand des aktuellen Zustandes entschieden, wie das aktuelle Byte zu verarbeiten ist (siehe Abbildung 3.3).

Wird im IDLE-Zustand auf Sendemodus gewechselt, so werden die vier Bytes der Erkennunssequenz gesendet. Intern wird für jedes Byte ein eigener Zustand verwendet. Danach folgt die Länge. Nun werden die so vorgegebene Anzahl Bytes an Daten gesendet. Der Übergang des Zustandes von IDLE zu Preambel erfolgt beim Einschalten des Sendemodus, falls sich im Sendepuffer ein Paket befindet. Danach erfolgt der Zustandsübergang automatisch nach dem Übermitteln des aktuellen Byte. Nach Übermitteln des letzten Datenbyte wird der Sendepuffer wieder gelöscht und in den IDLE Zustand gewechselt. Ist im IDLE Zustand kein Paket im Sendepuffer, wird das Radio in den Empfangsmodus geschaltet.

Im Empfangsmodus geschieht der Zustandsübergang durch Auswerten des aktuellen Byte.

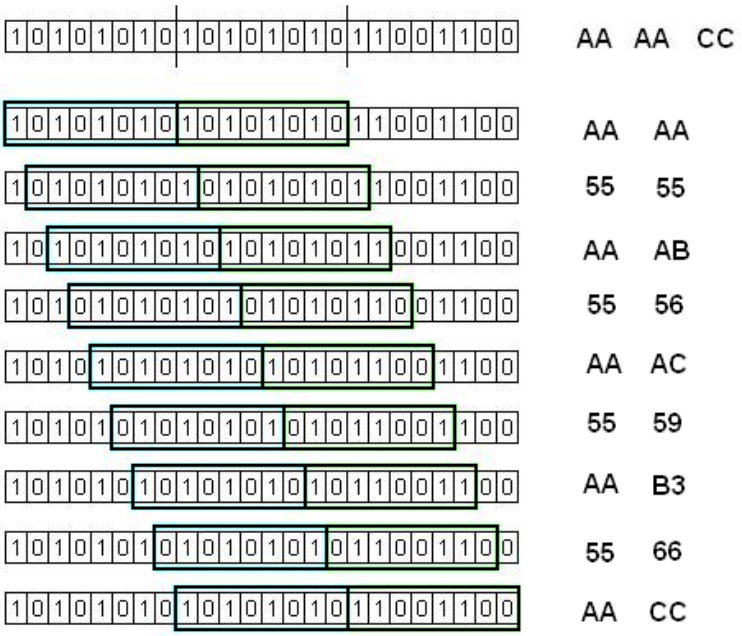

Abbildung 3.2: Byteverschiebung

Das erste Byte der Erkennungssequenz wechselt in selbigen Zustand, falls der Empfangspuffer leer ist. Auch hier wird intern für jedes Byte ein eigener Zustand verwendet. Kann die Verschiebung nicht erfolgreich berechnet werden oder stimmt das Prüfbyte nicht, wechselt der Treiber in den Zustand IDLE. Ansonsten wird als nächstes die Paketlänge emfangen, wobei hier noch überprüft werden muss, ob ein Paket dieser Länge in den Empfangspuffer passt. Falls nicht, wird ebenfalls abgebrochen. Danach werden die nun folgenden Bytes in den Empfangspuffer geschrieben. Nach Empfangen des letzten Byte, also beim Übergang zum IDLE Zustand, wird der Treiber-Prozess informiert, dass ein neues Paket im Empfangspuffer bereitsteht.

Um den Radiotreiber unabhängig von oberen Netzwerkschichten zu gestalten, wird mit einem Puffer gearbeitet. Bei Aufruf einer Sendefunktion wird das Paket in den Sendepuffer geschrieben und der Treiber angewiesen zu senden (Wechsel in den Sendemodus). Beim Empfangen wird, sobald das ganze Paket empfangen und in den Empfangspuffer kopiert wurde, die MAC-Schicht informiert, dass nun ein Paket angekommen und zum Lesen bereit ist. Diese kann nun durch Aufrufen einer Lesefunktion den Empfangspuffer auslesen. Nach diesem Auslesen wird der Empfangspuffer gelöscht und es kann ein weiteres Paket empfangen werden. Da der Treiber nur einen Puffer von einem Paket zu 255 Bytes verfügt, kann es zu Paketverlusten kommen, wenn die obere Netzwerkschicht die eingehenden Daten nicht schnell genug lesen kann oder in zu schneller Abfolge Daten senden will (siehe Kapitel 4).

3.4.3 Serielles Interface RS232

Die RS232 Schnittstelle ist bereits in der AVR Portierung von Contiki integriert. Eine Konstante für die Übertragungsgeschwindigkeit muss angepasst werden (Baudrate 57600), danach kann der Treiber geladen werden. Nun wird die Standardausgabe auf die serielle Schnittstelle gelegt,

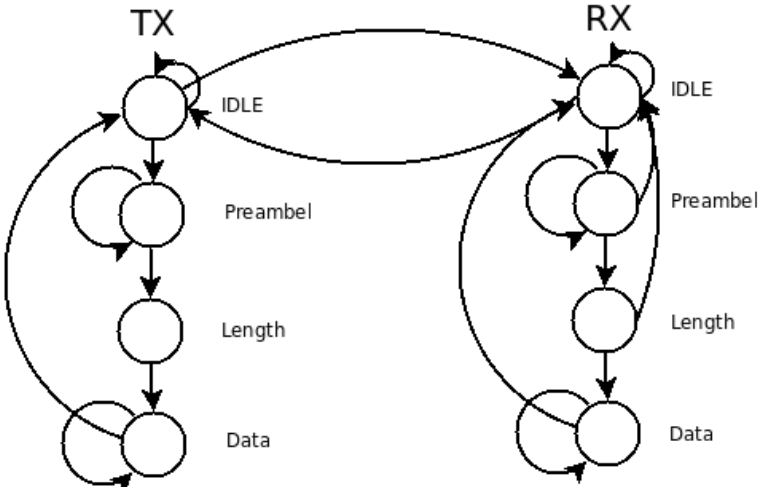

Abbildung 3.3: Zustandsdiagramm Radiotreiber

was es ermöglicht, mit der Verwendung von `printf` im Quellcode Ausgaben zu generieren.

3.4.4 LATCH

Der BTnode besitzt einen LATCH an welchem einerseits die LEDs, andererseits die Ein-/Aus-Schalter für das CC1000 Radio und Bluetooth angeschlossen sind. Der LATCH Eingang ist an die unteren 8 Bits des Adressbusses gekoppelt und kann über einen separaten Pin aktiviert werden. Da bei einer Änderung immer alle 8 Bits übergeben werden, aber meistens nur einzelne Bits geändet werden, muss der Treiber den aktuellen LATCH Zustand kennen und dann bei einer Änderung den resultierenden Zustand neu schreiben.

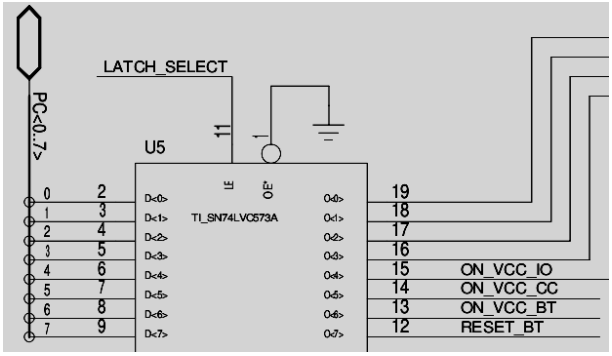

Abbildung 3.4: LATCH

3.4.5 LEDs

Contiki beinhaltet bereits Funktionen für die Steuerung der LEDs. Es muss jedoch eine hardwareabhängige Funktion implementiert werden, um ein LED ein und auszuschalten. Dies wird mithilfe des oben beschriebenen LATCHs erreicht. Ebenfalls werden die Farbdefinitionen angepasst, damit beispielsweise bei LED_GREEN auch wirklich das grüne LED leuchtet.

Kapitel 4

Evaluation

4.1 Radio

Um den Radio-Treiber zu testen, wird folgendes Szenario verwendet: Auf einem Senderknoten läuft ein Programm, welches von 0 bis 99 zählt und den jeweiligen Wert in einem Paket verpackt zu einem Empfängerknoten sendet.

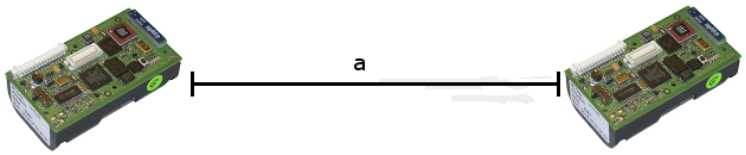

Abbildung 4.1: Versuchsanordnung mit zunehmendem Abstand a

Auf dem Empfängerknoten läuft ein Programm, welches auf eingehende Pakete hört und die so erhaltenen Daten auf dem seriellen Interface ausgibt. Die beiden BTnodes sind zusätzlich mit einer Antenne ausgestattet, um die Leistung zu verbessern. Es werden zwei Versuchsreihen durchgeführt. In der ersten Versuchsreihe wird die Anzahl der empfangene Pakete gemessen. Der Abstand zwischen den Sensorknoten beträgt wenige cm, die Sensorknoten stehen also direkt nebeneinander. Die Anzahl versendete Pakete pro Sekunde werden erhöht. In der zweiten Versuchsreihe werden 10 Pakete pro Sekunde, bei zunehmenden Abstand zwischen den Sensorknoten, gesendet (Versuchsanordnung in Abbildung 4.1). Es werden jeweils 20 Messungen durchgeführt, um eine grössere Datenmenge für die statistische Auswertung zu erhalten.

Abbildung 4.2 zeigt die Resultate der ersten Versuchsreihe. Es wird die prozentuale Anzahl der erfolgreich empfangenen Pakete bezüglich der Übertragungsrate dargestellt. Die Übertragungsgeschwindigkeit des CC1000 beträgt 76.8Kbaud, was bei Manchester Kodierung 38.4Kbit/s entspricht. Bei einer Paketgrösse von 64 Bits entspricht dies 612 Paketen pro Sekunde. Es ist zu erwarten, dass der tatsächliche Wert tiefer liegt, da bei jedem Paket das Radio zuerst auf Sendemodus geschaltet und nach der Übertragung wieder ausgeschaltet wird. Im Diagramm ist zu sehen dass die maximale Übertragungsrate mit dieser Versuchsanordnung etwas weniger als 300 Pakete pro Sekunde beträgt. Wird diese Rate überschritten, so kann jedes

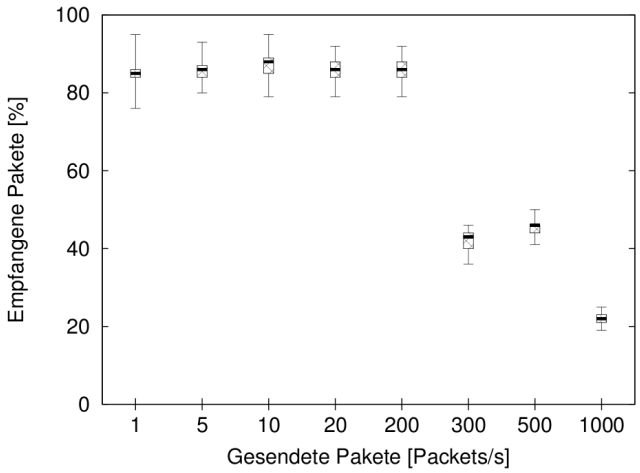

Abbildung 4.2: Paketverlust mit unterschiedlichen Übertragunsraten

zweite Paket nicht gesendet werden, da der Sendepuffer beim Sendeversuch bereits voll ist. Da der Versuch ohne Übertragungswiederholung durchgeführt wurde, gehen diese Pakete verloren. Wird die Rate weiter erhöht, gehen entsprechend mehr Pakete auf diese Weise verloren.

Abbildung 4.3 zeigt die Resultate der zweiten Versuchsreihe. Hier wird die prozentuale Anzahl erfolgreich empfangener Pakete bezüglich des Abstandes zwischen zwei Sensorknoten dargestellt. Messungen bis 30 Meter wurden im Gebäude durchgeführt, Messungen mit 50 und 70 Metern ausserhalb des Gebäudes. Die beiden Sensorknoten hatten dabei Sichtkontakt zueinander. Die Messung über 50 und 70 Metern war aufgrund der Witterungsbedingungen (Temperaturen unter 0°C) schwierig durchzuführen. Dies resultierte insbesondere bei der Messung über 70 Metern in einer etwas kleineren Messreihe. Allgemein kann aus den Messungen geschlossen werden, dass bis zu einer Distanz von ca. 50 Metern noch eine brauchbare Übertragung möglich ist. Bei grösseren Distanzen fällt die Rate der erfolgreich übertragenen Pakete markant ab. Dabei spielen sicherlich neben der Distanz auch externe Faktoren, wie störende Signale, eine wesentliche Rolle.

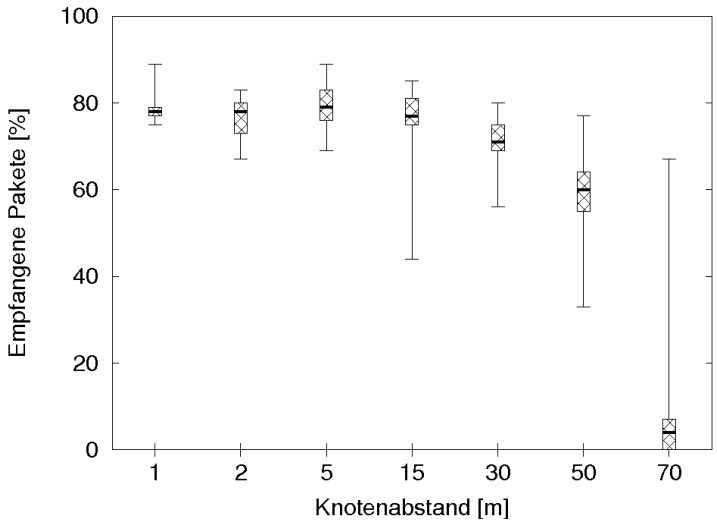

Abbildung 4.3: Paketverlust bei Übertragung mit unterschiedlichen Knotenabständen

Kapitel 5

Zusammenfassung

Ziel war es, eine funktionierenden Contiki Portierung für den BTnode zu entwickeln. Dazu wurden für die verschiedenen Hardwarekomponenten Treiber geschrieben und ins Contiki Betriebssystem eingebunden. Dabei fiel der Schwerpunkt auf den Radiotreiber, welcher danach mithilfe von Tests ausgewertet wurde. Die Implementierung beschränkte sich auf die Treiber, es wurde also am Contiki Kern nichts verändert. Als MAC Protokoll wurde nullmac verwendet welches in Contiki integriert ist. Mit der Portierung von Contiki für den BTnode wird ermöglicht, BTnodes in einem heterogenen Sensornetzes zu betreiben.
Bei der Auswertung wurde festgestellt, dass die Paketübertragung mit zusätzlich angebrachter Antenne für Distanzen von bis etwa 50 Metern relativ gut ist. Bei der Übertragungsrate liegt der in der Evaluation ermittelte Wert noch unterhalb (ca. 50%) der theoretisch möglichen Rate und könnte mit einem angepasstem MAC Protokoll sicherlich noch etwas gesteigert werden.

Literaturverzeichnis

[1] Gerald Wagenknecht, Markus Anwander, Torsten Braun, Thomas Staub, James Matheka, Simon Morgenthaler: MARWIS: A Management Architecture for Heterogeneous Wireless Sensor Networks, *6th International Conference on Wired/Wireless Internet Communications (WWIC'08)*

[2] http://www.sics.se/contiki/ *01.03.2010*

[3] http://www.btnode.ethz.ch *01.03.2010*

[4] http://www.ethernut.de/de/software.html *01.03.2010*

[5] http://www.tinyos.net/ *01.03.2010*

[6] Atmel ATmega128l Processor Manual

[7] CC1000 PRELIMINARY Datasheet *(rev. 2.1) 2002-04-19*

[8] Adam Dunkels , Oliver Schmidt, Thiemo Voigt , Muneeb Ali: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems *Swedish Institute of Computer Science*

[9] Adam Dunkels: Rime - A Lightweight Layered Communication Stack for Sensor Networks *Swedish Institute of Computer Science*